Introduction to R: Data Wrangling

Lisa Federer, Research Data Informationist

March 28, 2016

This course is designed to give you a simple and easy introduction to R, a programming language that can be used for data wrangling and processing, statistical analysis, visualization, and more. This handout will walk you through every step of today's class. Throughout the handout, you'll see the example code displayed like this:

```
> print(2 + 2)

[1] 4
```

The part that is in italics and preceded by the > symbol is the code that you will type. The part below it is the output you should expect to see. Sometimes code doesn't have any output to print; in that case, you'll just see the code and nothing else.

Also, sometimes the code is too long to fit on a single line of text in this handout. When that is the case, you will see the code split into separate lines, each starting with a +, like this:

```
> long_line_of_code <- c("Some really long code", "oh my gosh, how long is it going to be?",
+     "is it going to go on forever?", "I don't know, AGGGHHHHH",
+     "please, make it stop!")
```

When this is the case, do not insert any line breaks, extra spaces, or the plus sign - your code should be typed as one single line of code. Note that the default for your display in R Studio is not to wrap lines of text in your code, but you can turn this on by going to Tools > Global Options > Code Editing, and check the box next to "Soft-wrap R source files."

1 Getting Started

In this class, we'll be using RStudio, which is what's known as an *integrated development environment* (IDE). One nice shortcut that will be of help to you as you work in RStudio: tab completion. Hit tab after you type the first part of your command or name, and R will show you some options of what it things you could mean, and you can select one so you don't have to type out the whole word. RStudio also allows us to take some shortcuts that mean we don't have to code everything by hand. For example, we can use the navigation pane to set our working directory using the "Set as Working Directory" option under the "More" button. The working directory is the folder where R will look in if we give it a file name, and also where it will save things if we ask it to do so. We can also set our working directory by using the *setwd* command. Here, I've specified the folder on my computer where my files are, but you would use the path where your data files are stored.

```
> setwd("Z:/Data Services Workgroup/Data Instruction/R Classes/R Basics")
```

If you want to find out what your current working directory is, you can use `getwd`:

```
> getwd()

[1] "Z:/Data Services Workgroup/Data Instruction/R Classes/R Basics"
```

Next we'll read in our data. We'll be using a CSV file, but R can read almost any type of file. Let's check out the help text for this function first. Do so by typing `?read.csv` in the console. We don't need to use all of the arguments for the read.csv function. We'll just use the ones relevant to us, for which the default would not be what we want.

```
> master <- read.csv(file = "master.csv", header = TRUE)
```

You'll notice that no output has been printed with our code, but if you've been successful, you should now see `master` listed in your Global Environment pane.

1.1 Troubleshooting and Understanding Errors

When you first get started with R, expect to see lots of error messages! While you get used to the syntax and using R, it's natural that you'll make mistakes. Sometimes it's hard to figure out what your mistake is, but here are some helpful hints to troubleshooting some common errors.

```
> summary(Master)

Error in summary(Master) : object 'Master' not found
```

R has told me it can't find what I asked it to look for; in other words, I've told R to use some object that doesn't exist. If you see this error, check your spelling carefully and make sure you've used the correct capitalization - R is case sensitive.

R also gets confused if you use punctuation incorrectly, since it relies on puncutation marks for meaning. Beginners often leave out closing punctuation or have extra punctuation marks they don't need. RStudio is helpful in this regard because it will add closing punctuation when you type an opening punctuation. For example, if I type an open paranthesis, Rstudio puts a close paranthesis after it. Of course it's still possible to make mistakes, so check your code carefully.

Here is another command that will create an error (I can't even get it to run correctly in this documentation to make it print for you, but try it yourself.)

```
> master <- read.csv(file = "master.csv, header = TRUE)
```

Here I've left out the close quotation marks that should come after the .csv. When I run this line, a + appears after it in the console, indicating that R is expecting something else. Since I never closed my quotation marks, it thinks I have more to say. If you see the + appear in your console, click your cursor into the console and hit the ESCAPE key to interrupt the current command. You should see a > appear in your console window, which means R is ready for a command. Then fix your code and try again.

```
> master <- read.csv(file = "master.csv" header = TRUE)
```

R will get confused if you leave out punctuation or put extra punctuation in. Here, I accidentally left out the comma between my file argument and header argument, so R gives me an error message about an unepxected symbol. R isn't smart enough to tell you what is missing or extra, but it will tell you roughly where you should be looking for the error - it stops trying to run any code when it finds the error, so I know that my mistake is somewhere around the header argument.

```
> master <- read.csv([file = "master.csv", header = TRUE)
```

Sometimes R will be able to tell you what punctuation it found that it didn't like, as it does in this case. I have a random [, so R tells me it doesn't like that.

```
> master <- tread.csv(file = "master.csv", header = TRUE)

Error in try(master <- tread.csv(file = "master.csv", header = TRUE)) :
  could not find function "tread.csv"
```

If you make a spelling mistake in a function or when you're passing an argument, R will tell you that it couldn't find that function. Check your spelling and try again.

1.2 Getting to Know Your Data

Let's explore our data and see what we have. There are a few ways we can learn more about what our data contains. You can simply type the name of your data frame to have it display the whole thing, but if your dataset is very large, this wouldn't be very convenient. It might be better just to see a part of your data. The command `head()` will show just the first six observations.

```
> head(master)
```

```
   playerID birthYear birthMonth birthDay birthCountry birthState  birthCity
1 aardsda01      1981          12       27          USA         CO     Denver
2 aaronha01      1934           2        5          USA         AL     Mobile
3 aaronto01      1939           8        5          USA         AL     Mobile
4  aasedo01      1954           9        8          USA         CA     Orange
5  abadan01      1972           8       25          USA         FL Palm Beach
6  abadfe01      1985          12       17         D.R.  La Romana  La Romana
  deathYear deathMonth deathDay deathCountry deathState deathCity nameFirst
1        NA         NA       NA                                       David
2        NA         NA       NA                                        Hank
3      1984          8       16          USA         GA   Atlanta    Tommie
4        NA         NA       NA                                         Don
5        NA         NA       NA                                        Andy
6        NA         NA       NA                                     Fernando
  nameLast        nameGiven weight height bats throws     debut finalGame
1  Aardsma      David Allan    205     75    R      R  4/6/2004 9/28/2013
2    Aaron      Henry Louis    180     72    R      R 4/13/1954 10/3/1976
3    Aaron       Tommie Lee    190     75    R      R 4/10/1962 9/26/1971
4     Aase   Donald William    190     75    R      R 7/26/1977 10/3/1990
5     Abad    Fausto Andres    184     73    L      L 9/10/2001 4/13/2006
6     Abad Fernando Antonio    220     73    L      L 7/28/2010 9/27/2014
   retroID   bbrefID
1 aardd001 aardsda01
2 aaroh101 aaronha01
3 aarot101 aaronto01
4 aased001  aasedo01
5 abada001  abadan01
6 abadf001  abadfe01
```

Likewise, `tail()` will show just the last six.

```
> tail(master)
```

```
       playerID birthYear birthMonth birthDay birthCountry birthState
18584 zuninmi01      1991          3       25          USA         FL
18585 zupcibo01      1966          8       18          USA         PA
18586  zupofr01      1939          8       29          USA         CA
18587 zuvelpa01      1958         10       31          USA         CA
18588 zuverge01      1924          8       20          USA         MI
18589 zwilldu01      1888         11        2          USA         MO
          birthCity deathYear deathMonth deathDay deathCountry deathState
18584    Cape Coral        NA         NA       NA
18585    Pittsburgh        NA         NA       NA
18586 San Francisco      2005          3       25          USA         CA
18587     San Mateo        NA         NA       NA
18588       Holland      2014          9        8          USA         AZ
18589     St. Louis      1978          3       27          USA         CA
```

```
        deathCity nameFirst nameLast       nameGiven weight height bats throws
18584                  Mike   Zunino Michael Accorsi    220     74    R      R
18585                   Bob   Zupcic          Robert    220     76    R      R
18586   Burlingame     Frank     Zupo    Frank Joseph    182     71    L      R
18587                  Paul  Zuvella            Paul    173     72    R      R
18588        Tempe    George Zuverink          George    195     76    R      R
18589 La Crescenta     Dutch Zwilling Edward Harrison    160     66    L      L
         debut finalGame  retroID   bbrefID
18584 6/12/2013 9/28/2014 zunim001 zuninmi01
18585  9/7/1991  8/4/1994 zupcb001 zupcibo01
18586  7/1/1957  5/9/1961 zupof101  zupofr01
18587  9/4/1982  5/2/1991 zuvep001 zuvelpa01
18588 4/21/1951 6/15/1959 zuveg101 zuverge01
18589 8/14/1910 7/12/1916 zwild101 zwilldu01
```

If you'd like to see a different number of results, you can specify that by adding an additional argument to your head/tail function. For example, this will show the first 2 observations.

```
> head(master, n = 2)
```

```
   playerID birthYear birthMonth birthDay birthCountry birthState birthCity
1 aardsda01      1981         12       27          USA         CO    Denver
2 aaronha01      1934          2        5          USA         AL    Mobile
  deathYear deathMonth deathDay deathCountry deathState deathCity nameFirst
1        NA         NA       NA                                       David
2        NA         NA       NA                                        Hank
  nameLast   nameGiven weight height bats throws     debut finalGame  retroID
1  Aardsma David Allan    205     75    R      R  4/6/2004 9/28/2013 aardd001
2    Aaron Henry Louis    180     72    R      R 4/13/1954 10/3/1976 aaroh101
    bbrefID
1 aardsda01
2 aaronha01
```

We can get ask for a list of the variables or column names in our dataset, which is useful in case we need to double check the spelling or capitalization of our variables. This is also helpful because we can use it to find out which column number corresponds to which variable. We'll see later that we can refer to a column by its number or name.

```
> names(master)
```

```
 [1] "playerID"     "birthYear"    "birthMonth"   "birthDay"     "birthCountry"
 [6] "birthState"   "birthCity"    "deathYear"    "deathMonth"   "deathDay"
[11] "deathCountry" "deathState"   "deathCity"    "nameFirst"    "nameLast"
[16] "nameGiven"    "weight"       "height"       "bats"         "throws"
[21] "debut"        "finalGame"    "retroID"      "bbrefID"
```

We can also get some basic summary statistics about our data.

```
> summary(master)
```

```
      playerID       birthYear      birthMonth        birthDay
 aardsda01:    1   Min.   :1820   Min.   : 1.000   Min.   : 1.00
 aaronha01:    1   1st Qu.:1894   1st Qu.: 4.000   1st Qu.: 8.00
 aaronto01:    1   Median :1935   Median : 7.000   Median :16.00
 aasedo01 :    1   Mean   :1930   Mean   : 6.626   Mean   :15.62
 abadan01 :    1   3rd Qu.:1967   3rd Qu.:10.000   3rd Qu.:23.00
```

```
 abadfe01 :    1   Max.   :1994   Max.   :12.000   Max.   :31.00
 (Other)  :18583   NA's   :145    NA's   :315      NA's   :472
    birthCountry     birthState          birthCity       deathYear
 USA      :16322   CA     : 2115   Chicago     :  376   Min.   :1872
 D.R.     :  619   PA     : 1414   Philadelphia:  356   1st Qu.:1941
 Venezuela:  321   NY     : 1202   St. Louis   :  296   Median :1966
 P.R.     :  246   IL     : 1051   New York    :  267   Mean   :1963
 CAN      :  244   OH     : 1030   Brooklyn    :  240   3rd Qu.:1988
 Cuba     :  191   TX     :  873   Los Angeles :  228   Max.   :2014
 (Other)  :  646   (Other):10904   (Other)     :16826   NA's   :9364
   deathMonth        deathDay        deathCountry    deathState
 Min.   : 1.000   Min.   : 1.00            :9366          :9420
 1st Qu.: 3.000   1st Qu.: 8.00   USA      :9017   CA     :1059
 Median : 6.000   Median :15.00   CAN      :  63   PA     : 792
 Mean   : 6.486   Mean   :15.55   Cuba     :  26   NY     : 692
 3rd Qu.:10.000   3rd Qu.:23.00   Mexico   :  23   OH     : 576
 Max.   :12.000   Max.   :31.00   Venezuela:  21   FL     : 545
 NA's   :9365     NA's   :9366    (Other)  :  73   (Other):5505
        deathCity      nameFirst         nameLast             nameGiven
             :9370   Bill   :  549   Smith   :  150   John Joseph   :   74
 Philadelphia: 238   John   :  480   Johnson :  110   John          :   56
 Chicago     : 181   Jim    :  443   Jones   :   97   William Henry :   53
 St. Louis   : 175   Mike   :  429   Brown   :   89   William Joseph:   48
 Los Angeles : 136   Joe    :  396   Miller  :   88   William       :   44
 New York    : 124   Bob    :  341   Williams:   76   Michael Joseph:   42
 (Other)     :8365   (Other):15951   (Other) :17979   (Other)       :18272
     weight          height      bats      throws           debut
 Min.   : 65.0   Min.   :43.00    : 1190    :  979            :  190
 1st Qu.:170.0   1st Qu.:71.00   B: 1150   L: 3542   1884-05-01:   36
 Median :185.0   Median :72.00   L: 4814   R:14068   1879-05-01:   26
 Mean   :185.6   Mean   :72.24   R:11435             1882-05-02:   26
 3rd Qu.:197.0   3rd Qu.:74.00                       1884-04-17:   25
 Max.   :320.0   Max.   :83.00                       4/13/1954 :   24
 NA's   :872     NA's   :809                         (Other)   :18262
     finalGame          retroID           bbrefID
 9/28/2014:  464            :   54            :    1
 9/27/2014:  213   aardd001:    1   aardsda01:    1
           :  190   aaroh101:    1   aaronha01:    1
 9/26/2014:  105   aarot101:    1   aaronto01:    1
 9/25/2014:   55   aased001:    1   aasedo01 :    1
 9/29/2013:   54   abada001:    1   abadan01 :    1
 (Other)  :17508   (Other) :18530   (Other)  :18583
```

Different types of stats are shown depending on the type of variable. For example, some of the variables, like weight, give us the mean, median, and other summary stats. Other variables, like birthState, give a count of how many items are in each category. Let's explore why this is by figuring out the class of these variables. A variable is referred to by using the date frame name, the $ and then the variable name.

```
> class(master$birthState)

[1] "factor"

> class(master$weight)

[1] "integer"
```

Another way we can get that information is by asking R to show us our data's structure.

```
> str(master)

'data.frame':        18589 obs. of  24 variables:
 $ playerID    : Factor w/ 18589 levels "aardsda01","aaronha01",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ birthYear   : int  1981 1934 1939 1954 1972 1985 1854 1877 1869 1866 ...
 $ birthMonth  : int  12 2 8 9 8 12 11 4 11 10 ...
 $ birthDay    : int  27 5 5 8 25 17 4 15 11 14 ...
 $ birthCountry: Factor w/ 53 levels "","Afghanistan",..: 50 50 50 50 50 18 50 50 50 50 ...
 $ birthState  : Factor w/ 239 levels "","AB","Aberdeen",..: 42 6 6 28 66 104 168 168 222 144 ...
 $ birthCity   : Factor w/ 4683 levels "","Aberdeen",..: 1085 2698 2698 3071 3137 2195 3256 2273 1328 13
 $ deathYear   : int  NA NA 1984 NA NA NA 1905 1957 1962 1926 ...
 $ deathMonth  : int  NA NA 8 NA NA NA 5 1 6 4 ...
 $ deathDay    : int  NA NA 16 NA NA NA 17 6 11 27 ...
 $ deathCountry: Factor w/ 23 levels "","American Samoa",..: 1 1 21 1 1 1 21 21 21 21 ...
 $ deathState  : Factor w/ 93 levels "","AB","AK","AL",..: 1 1 26 1 1 1 57 25 88 12 ...
 $ deathCity   : Factor w/ 2527 levels "","Aberdeen",..: 1 1 90 1 1 1 1716 750 457 1960 ...
 $ nameFirst   : Factor w/ 2268 levels "","A. J.","Aaron",..: 505 906 2049 588 72 764 1133 638 156 332 .
 $ nameLast    : Factor w/ 9605 levels "Aardsma","Aaron",..: 1 2 2 3 4 4 5 6 7 7 ...
 $ nameGiven   : Factor w/ 12264 levels "","A. Harry",..: 2451 5033 11261 2837 3678 3712 6527 3149 937 1
 $ weight      : int  205 180 190 190 184 220 192 170 175 169 ...
 $ height      : int  75 72 75 75 73 73 72 71 71 68 ...
 $ bats        : Factor w/ 4 levels "","B","L","R": 4 4 4 4 3 3 4 4 4 3 ...
 $ throws      : Factor w/ 3 levels "","L","R": 3 3 3 3 2 2 3 3 3 2 ...
 $ debut       : Factor w/ 9910 levels "","10/1/1900",..: 2806 1555 1411 6077 7976 6177 233 1259 1002 10
 $ finalGame   : Factor w/ 8924 levels "","10/1/1906",..: 8477 148 8255 155 1526 8384 349 7390 1324 1356
 $ retroID     : Factor w/ 18536 levels "","aardd001",..: 2 3 4 5 6 7 8 9 10 11 ...
 $ bbrefID     : Factor w/ 18589 levels "","aardsda01",..: 2 3 4 5 6 7 8 9 10 11 ...
```

2 Processing and Working with Data

In this section we'll look at how to use R to easily do some basic processing and cleaning of our data to make it easier to work with. This data wrangling step is of huge importance to any sort of data analysis and is typically about 80% of any data science process.

One of the things we might want to do is to take two different datasets and combine them using some unique ID that both datasets share. R is very good at this! Let's read in another dataset about our baseball players. Now we will have two different data frames to work with.

```
> batting <- read.csv(file = "batting.csv", header = TRUE)
```

2.1 Subsetting

It's often desirable to use just a subset of data, particularly with a large dataset. Before we try to merge our batting data with our master table, we'll create some new data frames that contain just parts of our larger dataset.

Let's create a new data frame using just the data from 2014. There are a number of ways we could do this. First, we could use the subset function to select observations that exactly meet a specific criteria, in this case, only those from the year 2014.

```
> batting2014 <- subset(batting, yearID == "2014")
```

Since year is an integer variable we can also use operators for subsetting, like selecting all observations that are greater than or equal to 2014. We can also use Boolean operators, via the ampersand (for AND) and the pipe sign (for OR) to create complex conditions.

```
> batting2014 <- subset(batting, yearID >= 2014)
> frequent_batting2014 <- subset(batting, yearID >= 2014 & AB >
+     20)
```

We can even nest conditions with parentheses to get really specific, like here, where we've selected just those players who had more than 50 runs OR more than 100 homers in the year 2014.

```
> best_batting2014 <- subset(batting, yearID >= 2014 & (R > 50 |
+     HR >= 25))
```

Another way we can subset data is by referring to specific rows, columns, or both using the []. When something is inside square brackets, R interprets it as [row, column]. We can refer to just rows, just columns, or both, using the number of the row/column. We can also ask for a set of rows or columns, referred to as a slice, using the :. For example, here I'm going to create a new data frame with just the first 5 observations in my batting data frame. Because I put nothing after the comma, it will take all the columns for those first 5 observations.

```
> first5 <- batting[1:5, ]
```

Conversely, if I put nothing before the comma, I'll have all of the rows, but just the first two columns of all of those observations.

```
> player.year <- batting[, 1:2]
```

I can also remove things I don't want. Here I'm making a new dataset with everything except the 3rd column.

```
> no_3rd <- batting[, -3]
```

I can also specify both row and column information if I want.

```
> new_batting <- batting[1:100, -3]
```

I can use the : to take all the rows or columns in a range, but I can also use *c()* (fun fact: the c stands for concatenate, which means to link things together in a series!) to refer to some specific rows or columns, such as here, where I'm taking rows 1-100 and 400-425 of columns 1, 2, and 5.

```
> new_batting <- batting[c(1:100, 400:425), c(1, 2, 5)]
```

I can even have R generate a random sample for me. Here, I've requested it to look through all the rows in my batting dataset and choose 50 random observations.

```
> random.batting <- batting[sample(1:nrow(batting), 50), ]
```

2.2 Renaming Variables

One of the things you may have noticed about our dataset is that our variable names aren't very descriptive. We have 22 variables, but most of them are just a letter, which isn't helpful to us if we don't have the codebook in front of us. Let's rename some of our variables. As is so often the case with R, there are many ways to do this! One way is to tell R to take look through the variable names for the one that is currently called R and rename it as runs.

```
> names(batting2014)[names(batting2014) == "R"] <- "runs"
```

I can also use the number of the column to tell R which variable to rename.

```
> names(batting2014)[6] <- "games"
```

Another, even easier, way to do this relies on a package called plyr.

2.3 Installing and Working with Packages

R has quite a bit of built-in functionality, but there are also many, many free packages that allow us to do other things that aren't part of the base R functionality. We can download and install these packages, then tell R which packages you want to use, in order to accomplish lots of different things. Working with packages is one thing that's much easier with RStudio. We can interact with the Packages tab and RStudio will figure out the right code to run for us. Using the install button, install the package plyr now.

once you've installed a package, you need to tell RStudio when you want to use it. You can do that by checking the box next to its name in the Packages tab list, or you can do so by loading it using the `library()` function.

```
> library(plyr)
```

Now we can use the handy rename function that's part of plyr. Simply enter the names of all the variables you'd like to rename. Notice the variable "X2B." Variables can't start with numbers, so when R read our CSV file in, it changed the variable name "2B" to "X2B"

```
> batting2014 <- rename(batting2014, c(AB = "at_bats", H = "hits",
+     X2B = "doubles"))
```

Let's check out our new names for our batting2014 dataset

```
> names(batting2014)
```

```
 [1] "playerID" "yearID"   "stint"    "teamID"   "lgID"     "games"
 [7] "at_bats"  "runs"     "hits"     "doubles"  "X3B"      "HR"
[13] "RBI"      "SB"       "CS"       "BB"       "SO"       "IBB"
[19] "HBP"      "SH"       "SF"       "GIDP"
```

2.4 Merging Datasets

Now that we've gotten our batting dataset renamed, let's merge our master dataset with player info, with our batting dataset. We can merge two datasets using the `merge()` function as long as the two datasets share at least one common column name. Let's see if they do. Both the master and the batting datasets have a variable called playerID, which is a unique identifier for each player. I'm going to create a new data frame by merging my two data frames.

```
> full_data <- merge(batting2014, master, by = "playerID")
```

Notice that our new data frame has 1435 observations. Each of our players has been matched with his master data. But not every player in the master list of 18,000+ players is in our 2014 batting dataset. What if we wanted to still include all players in the new data frame, even if we didn't have their batting details? In the merge function, our first dataframe is considered X, and the second Y. So if we use all.y = TRUE, R will add every single observation from the master list, even if there is no corresponding match in batting.

```
> full_data_all <- merge(batting2014, master, by = "playerID",
+     all.y = TRUE)
```

2.5 Recoding Data, Creating New Variables, and Changing Data Types

Sometimes it's desirable to recode data (for example, changing all No responses to 0 and Yes responses to 1), create new variables based on existing data, or adjust the class of data (for example, changing a factor to an ordered factor).

Let's suppose we'd like to recode our data to use the full state name instead of the state abbreviation in our birth state column. We could do this for every state, but we'll just do a couple here so you get the idea. The revalue function we'll use here is part of the plyr package.

```
> full_data$birthState <- revalue(full_data$birthState, c(TX = "Texas",
+     CA = "California"))
```

If we look at our summary of our birth state variable now, we see that we have Texas and California spelled out, instead of abbreviated. (Note that I've included the maxsum argument here - this allows me to specify that I would only like to see the first n observations, rather than the entire summary for this variable.)

```
> summary(full_data$birthState, maxsum = 5)
```

```
California      Texas         FL              (Other)
       223        119         91         52        950
```

Sometimes it's useful to create a new variable based on some existing data. For example, we could make a new ordinal variable called based on the height variable, categorizing our players as short, average, or tall. We'll do this one category at a time.

```
> full_data$rel_height[full_data$height < 70] <- "short"
> full_data$rel_height[full_data$height >= 70 & full_data$height <=
+     75] <- "average"
> full_data$rel_height[full_data$height > 75] <- "tall"
```

We have our new variable now, but we have a problem:

```
> class(full_data$rel_height)
```

```
[1] "character"
```

Our new variable was created as a character variable, because R saw that we had entered letters and made the assumption that this would be a character variable. It would be better to create a factor variable from this, because this is actually an ordinal variable. If R considers it a character variable, it's just seeing the data as a bunch of letters with no meaning. Actually, there is more meaning to our variable than that, so we're going to turn out data into an ordered factor, so that R knows that short is shorter than average, which is shorter than tall.

To do this, first we need to convert the variable to a factor variable.

```
> full_data$rel_height <- as.factor(full_data$rel_height)
```

At this point, it lacks ordering, so there is no understood relationship - the three categories are just presumed to be three separate groups. One way we can tell that R doesn't know the order of our factor levels yet is by having it show us a table of values. R arranges the table in alphabetical order.

```
> table(full_data$rel_height)
```

```
average   short    tall
   1055      62     318
```

Now that this variable is a factor, we can add ordering to the variable to introduce the relationship between the different groups. This would be important for certain types of analyses (for example, if you were analyzing Likert scale data using some packages) and would also ensure that our data points are arranged correctly if we want to create a visualization of our data.

```
> full_data$rel_height <- factor(full_data$rel_height, levels = c("short",
+     "average", "tall"))
```

Now we can see our data are in the correct order.

```
> table(full_data$rel_height)
```

```
  short average    tall
     62    1055     318
```

We can also create new variables that transform existing variables by doing some math on it. For example, right now we have our weight in pounds, but perhaps we'd like to change that to kilograms. I'll simply multiply my weight column by the multiplier to convert between pounds and kilograms and assign the output to a new column.

```
> full_data$wt_kg <- full_data$weight * 0.453592
```

2.6 Dealing with NAs

Sometimes we are missing data for whatever reason. Typically this is handled in R by using NA. In some cases, this can cause some issues for us. For example, suppose we want to calculate the mean number of runs for all our players.

```
> mean(full_data$runs)

[1] 13.77073
```

Because we have some NA values in our runs variable, the mean is also NA. However, we can tell R to ignore the NAs when it's doing the calculation and just give us the mean of the existing data in this column.

```
> mean(full_data$runs, na.rm = TRUE)

[1] 13.77073
```

Sometimes it's desirable to create a subset of our data that only contains observations for which we have all the data, in other words, no NAs. We can do this using the na.omit() function. This function will remove all observations that are not complete, that is to say, those that are missing data in any variable. Keep in mind that this will delete the *entire* row if it finds even one NA in that row.

```
> no_na <- na.omit(batting)
```

A more conservative approach that might be acceptable is just to remove those observations that have NAs in key variables. For example, maybe I want only those players for whom I have a value in the runs variable, and I want to discard all the players whose value for runs is NA. I can do that by using !is.na. The exclamation mark stands for NOT - in other words, I'm asking for everything that is NOT NA.

```
> has_runs <- full_data[!is.na(full_data$runs), ]
```

As a note, we could have also specified an NA value when we were reading in our data file. For example, suppose when I created my CSV, I used the code 999 to indicate NA. I could pass an additional argument to the read.csv function telling it to look for all 999s and replace that with NA when it's reading in my data.

```
> new_data <- read.csv(file = "master.csv", na.strings = "999")
```

2.7 Working with Tables

So far we've primarily dealt with our data in the form of a data frame. Another class of data that we might want to use is tables. Some statistical tests require that data be in a table. Tables are also nice when we want to get a feel for how our data are distributed. For example, let's suppose we'd like to create a table of height and whether a player is left or right-handed. We first need to make a minor modification to our data to remove empty levels from our "throws" variable, which tells us whether a player is left or right handed.

```
> full_data$throws <- factor(full_data$throws)
```

To create our table, we simply tell R what variables we want included.

```
> handedtbl <- table(full_data$throws, full_data$rel_height)
> handedtbl

    short average tall
  L     9     234   78
  R    53     821  240
```

The order in which we pass our variables to the table function determines which are columns and which are rows. If we want to transpose our table, we'd simply change the order we pass our variable names.

```
> handedtbl <- table(full_data$rel_height, full_data$throws)
> handedtbl
```

```
           L   R
  short    9  53
  average 234 821
  tall     78 240
```

We can also create a proportion table - rather than giving us the raw values, the proportions are calculated. We can specify whether we want proportions calculated across rows (using margin = 1) or columns (using margin = 2).

```
> prop.handed <- prop.table(handedtbl, margin = 1)
> prop.handed

                 L         R
  short   0.1451613 0.8548387
  average 0.2218009 0.7781991
  tall    0.2452830 0.7547170

> prop.handed <- prop.table(handedtbl, margin = 2)
> prop.handed

                   L          R
  short   0.02803738 0.04757630
  average 0.72897196 0.73698384
  tall    0.24299065 0.21543986
```

3 Iteration and the Apply Functions

One of the great benefits of R is that we can use it to automate a lot of the work we need to do to our data. We don't have to manually change or analyze our data column by column or row by row - we can write one set of code and iterate it many times over a set of data.

If you're familiar with other programming languages, you may be used to doing this type of iteration using what's called a for loop. Though it's possible to write for loops in R, this is somewhat frowned upon because it takes longer and is less efficient than using some other methods of iteration. We'll focus on these other methods here.

3.1 Apply

There are several different apply functions that take your instructions and apply them over a set of data. The first, and most basic, is the apply() function, which will do your function over set of columns or rows. We'll specify whether we want columns or rows (or both) in the same way we did above when we referred to "margins" in our tables. For example, suppose I'd like to find the mean for each of my columns in my data frame.

The apply function takes 3 arguments:

- the data of interest
- the margin I want (1 for row, 2 for column, c(1,2) for both)
- the function I want to apply (in this case, mean)

Another note about how I've done this here - remember how the mean function will return NA if our data has any NA values? I want to remove those NA values and get the mean for the non-NA data. I'm also only interested in the columns containing numeric values, which in our data frame are columns 6-22. I don't have to create a new data frame to do this - I can just nest my functions and put it all in one line of code.

```
> apply(na.omit(full_data[, 6:22]), 2, mean)
```

```
      games     at_bats        runs        hits     doubles         X3B
 48.4766551 115.4104530  13.7707317  28.9860627   5.6703833   0.5916376
         HR         RBI          SB          CS          BB          SO
  2.9170732  13.0627178   1.9261324   0.7212544   9.7700348  26.0912892
        IBB         HBP          SH          SF        GIDP
  0.6864111   1.1512195   0.9358885   0.8898955   2.5149826
```

Now we have the mean for every variable we're interested in!

3.2 By

Let's suppose we're interested in those same means, but this time, we want to break it up by groups. Our data include which league the player is in, the American League or the National League, and I want to see if there are differences in the means for these two leagues. We can do so using the by() function.

The by function takes 3 arguments:

- the data I'm interested in looking at (we're going to use the same data as we just used for apply)
- the variable I want to group on (in this case, lgID)
- the function I want to apply (I'm using colMeans now instead of means because I'm no longer given that margins option to specify how to apply the function and therefore I need to give more specific instructions)

```
> by(na.omit(full_data[, 6:22]), full_data$lgID, colMeans)

full_data$lgID: AA
NULL
------------------------------------------------------------
full_data$lgID: AL
      games     at_bats        runs        hits     doubles         X3B
 46.6970509 111.4249330  13.6206434  28.1943700   5.5388740   0.5227882
         HR         RBI          SB          CS          BB          SO
  2.8967828  12.9168901   1.8686327   0.6595174   9.5053619  24.4571046
        IBB         HBP          SH          SF        GIDP
  0.6313673   1.1126005   0.6126005   0.9101877   2.4852547
------------------------------------------------------------
full_data$lgID: FL
NULL
------------------------------------------------------------
full_data$lgID: NL
      games     at_bats        runs        hits     doubles         X3B
 50.4034833 119.7256894  13.9332366  29.8432511   5.8127721   0.6661829
         HR         RBI          SB          CS          BB          SO
  2.9390421  13.2206096   1.9883890   0.7880987  10.0566038  27.8606676
        IBB         HBP          SH          SF        GIDP
  0.7460087   1.1930334   1.2859216   0.8679245   2.5471698
------------------------------------------------------------
full_data$lgID: PL
NULL
------------------------------------------------------------
full_data$lgID: UA
NULL
```

3.3 The Other Apply Functions

There are a whole bunch of other apply-like functions that can be used depending on what type of data you're putting into the function and what kind of data you hope to get out. For example, lapply returns a list, thus the "l" in the name, while mapply lets you apply to the funciton to multiple arguments, thus the "m." We won't get into these here, but be aware that there are lots of options available depending on the type of data you're working with.

4 Restructuring and Reorganizing Data

We've now dealt with a variety of functions that let us modify and process our data. Now, we'll deal with some ways of restructuring and reorganizing our data.

4.1 Sorting Data

First, let's look at how to sort a data frame. Right now, our data frame is sorted alphabetically by player ID, but what if I'd like to sort my data by weight? We can sort by any variable using the order function. (By the way, if you were saving this, you would assign it a name using the assignment operator, but here, I'm not saving this as a new data frame and I'm wrapping it in the head() function so that the whole several thousand observations long data frame isn't printed in your handout.)

```
> head(full_data[order(full_data$weight), ])
```

```
      playerID yearID stint teamID lgID games at_bats runs hits doubles X3B HR
34   amarial01   2014     1    SDN   NL   148     423   39  101      13   2  5
582  herredi01   2014     1    NYN   NL    18      59    6   13       0   1  3
1119 rojasmi02   2014     1    LAN   NL    85     149   16   27       3   0  1
1178 sardilu01   2014     1    TEX   AL    43     115   12   30       6   0  0
110  bettsmo01   2014     1    BOS   AL    52     189   34   55      12   1  5
333   diazjo02   2014     1    TOR   AL    23      38    3    6       1   0  0
     RBI SB CS BB SO IBB HBP SH SF GIDP birthYear birthMonth birthDay
34    40 12  1 29 69   5   1  8  5    6      1989          4        6
582   11  0  0  7 17   0   0  0  0    3      1994          3        3
1119   9  0  0 10 28   1   2  1  0    5      1989          2       24
1178   8  5  1  5 21   0   2  3  0    5      1993          5       16
110   18  7  3 21 31   0   2  1  0    2      1992         10        7
333    4  1  0  3 14   0   2  2  0    1      1985          4       10
     birthCountry birthState   birthCity deathYear deathMonth deathDay
34      Venezuela Anzoategui   Barcelona        NA         NA       NA
582      Colombia    Bolivar   Cartagena        NA         NA       NA
1119    Venezuela    Miranda  Los Teques        NA         NA       NA
1178    Venezuela    Bolivar       Upata        NA         NA       NA
110           USA         TN   Nashville        NA         NA       NA
333           USA         FL Miami Beach        NA         NA       NA
     deathCountry deathState deathCity nameFirst nameLast      nameGiven weight
34                                      Alexi Amarista     Alexi Jose    150
582                                    Dilson  Herrera    Dilson Jose    150
1119                                   Miguel    Rojas   Miguel Elias    150
1178                                     Luis Sardinas Luis Alexander    150
110                                    Mookie    Betts    Markus Lynn    155
333                                  Jonathan     Diaz       Jonathan    155
     height bats throws     debut finalGame  retroID   bbrefID rel_height
34       66    L      R 4/26/2011 9/28/2014 amara001 amarial01      short
582      70    R      R 8/29/2014 9/20/2014 herrd002 herredi01    average
1119     72    R      R  6/6/2014 9/28/2014 rojam002 rojasmi02    average
```

```
1178     73    B      R 4/20/2014 9/28/2014 sardl001 sardilu01    average
110      69    R      R 6/29/2014 9/28/2014 bettm001 bettsmo01      short
333      69    R      R 6/29/2013 5/12/2014 diazj004  diazjo02      short
        wt_kg
34   68.03880
582  68.03880
1119 68.03880
1178 68.03880
110  70.30676
333  70.30676
```

By default, the sort will be done ascending, but I can also switch to a descending sort by passing the decreasing argument to the function.

```
> head(full_data[order(full_data$weight, decreasing = TRUE), ])
```

```
     playerID yearID stint teamID lgID games at_bats runs hits doubles X3B HR
334  diazju03   2014     1    CIN   NL    36       0    0    0       0   0  0
150 broxtjo01   2014     1    CIN   NL    51       0    0    0       0   0  0
151 broxtjo01   2014     2    MIL   NL    11       0    0    0       0   0  0
262 colonba01   2014     1    NYN   NL    31      62    3    2       1   0  0
362  dunnad01   2014     1    CHA   AL   106     363   43   80      17   0 20
363  dunnad01   2014     2    OAK   AL    25      66    6   14       1   0  2
    RBI SB CS BB  SO IBB HBP SH SF GIDP birthYear birthMonth birthDay
334   0  0  0  0   0   0   0  0  0    0      1984          2       27
150   0  0  0  0   0   0   0  0  0    0      1984          6       16
151   0  0  0  0   0   0   0  0  0    0      1984          6       16
262   0  0  0  0  33   0   0  7  0    0      1973          5       24
362  54  1  1 65 132   5   3  0  4    5      1979         11        9
363  10  0  0  6  27   0   4  0  0    0      1979         11        9
    birthCountry   birthState birthCity deathYear deathMonth deathDay
334         D.R.    La Romana La Romana        NA         NA       NA
150          USA           GA   Augusta        NA         NA       NA
151          USA           GA   Augusta        NA         NA       NA
262         D.R. Puerto Plata  Altamira        NA         NA       NA
362          USA        Texas   Houston        NA         NA       NA
363          USA        Texas   Houston        NA         NA       NA
    deathCountry deathState deathCity nameFirst nameLast    nameGiven weight
334                                        Jumbo     Diaz  Jose Rafael    315
150                                     Jonathan  Broxton Jonathan Roy    295
151                                     Jonathan  Broxton Jonathan Roy    295
262                                      Bartolo    Colon      Bartolo    285
362                                         Adam     Dunn    Adam Troy    285
363                                         Adam     Dunn    Adam Troy    285
    height bats throws     debut finalGame  retroID   bbrefID rel_height
334     76    R      R 6/20/2014 9/27/2014 diazj005  diazju03       tall
150     76    R      R 7/29/2005 9/27/2014 broxj001 broxtjo01       tall
151     76    R      R 7/29/2005 9/27/2014 broxj001 broxtjo01       tall
262     71    R      R  4/4/1997 9/28/2014 colob001 colonba01    average
362     78    L      R 7/20/2001 9/28/2014 dunna001  dunnad01       tall
363     78    L      R 7/20/2001 9/28/2014 dunna001  dunnad01       tall
      wt_kg
334 142.8815
150 133.8096
151 133.8096
```

```
262 129.2737
362 129.2737
363 129.2737
```

I can even sort by two variables if I want. For example, say I want to sort by birth year, and then in addition to that, sort by month. For that matter, I can even throw birth day in as well. The order in which I pass the variables to the order function is the order in which the data will be sorted.

```
> head(full_data[order(full_data$birthYear, full_data$birthMonth,
+     full_data$birthDay), ])

      playerID yearID stint teamID lgID games at_bats runs hits doubles X3B HR
473  giambja01   2014     1    CLE   AL    26      60    3    8       2   0  2
624  ibanera01   2014     1    LAA   AL    57     166   16   26       5   2  3
625  ibanera01   2014     2    KCA   AL    33      80    7   15       3   1  2
564  hawkila01   2014     1    COL   NL    57       1    0    0       0   0  0
262  colonba01   2014     1    NYN   NL    31      62    3    2       1   0  0
1276 suzukic01   2014     1    NYA   AL   143     359   42  102      13   2  1
     RBI SB CS BB SO IBB HBP SH SF GIDP birthYear birthMonth birthDay
473    5  0  0  9 12   2   1  0  0    3      1971          1        8
624   21  3  2 23 43   0   0  0  1    1      1972          6        2
625    5  0  0 10 16   0   0  0  0    1      1972          6        2
564    0  0  0  0  1   0   0  0  0    0      1972         12       21
262    0  0  0  0 33   0   0  7  0    0      1973          5       24
1276  22 15  3 21 68   1   1  2  2    3      1973         10       22
     birthCountry   birthState         birthCity deathYear deathMonth deathDay
473           USA   California       West Covina        NA         NA       NA
624           USA           NY          New York        NA         NA       NA
625           USA           NY          New York        NA         NA       NA
564           USA           IN              Gary        NA         NA       NA
262          D.R. Puerto Plata          Altamira        NA         NA       NA
1276        Japan        Aichi Nichi Kasugai-gun        NA         NA       NA
     deathCountry deathState deathCity nameFirst nameLast     nameGiven weight
473                                        Jason   Giambi Jason Gilbert    240
624                                         Raul   Ibanez   Raul Javier    225
625                                         Raul   Ibanez   Raul Javier    225
564                                       LaTroy  Hawkins        LaTroy    220
262                                      Bartolo    Colon       Bartolo    285
1276                                      Ichiro   Suzuki        Ichiro    170
     height bats throws     debut finalGame  retroID   bbrefID rel_height
473      75    L      R  5/8/1995 9/27/2014 giamj001 giambja01    average
624      74    L      R  8/1/1996 9/28/2014 ibanr001 ibanera01    average
625      74    L      R  8/1/1996 9/28/2014 ibanr001 ibanera01    average
564      77    R      R 4/29/1995 9/27/2014 hawkl001 hawkila01       tall
262      71    R      R  4/4/1997 9/28/2014 colob001 colonba01    average
1276     71    L      R  4/2/2001 9/28/2014 suzui001 suzukic01    average
        wt_kg
473  108.86208
624  102.05820
625  102.05820
564   99.79024
262  129.27372
1276  77.11064
```

4.2 Transforming and Summarizing Data

We looked at the by function, which allowed us to apply a function to data by group to get some summary data for each of our groups. The ddply function in the plyr package can also be useful for doing this. Let's suppose I'd like to compile some team stats by finding all the players for each team, and adding up their home runs and runs.

The ddply function needs a few things:

- the data (I'll use my batting data frame here)
- the variable(s) I want to group on (here I'm using both team and year, so I'll get the stats by team for each year)
- I'm specifying that I want to summarize - this means that a new, condensed data frame will be created with just the data I'm specifying here
- what I want to call my new variables and how I want them to be calculated. In this example, I'm getting the total runs and homeruns per team, per year by using the sum function, as well as the mean homeruns and runs per player, per year, per team. I'm also specifying that I want to omit NAs just in case I have any missing data

```
> team_summary <- ddply(batting, c("teamID", "yearID"), summarize,
+     total_homeruns = sum(HR, na.rm = TRUE), total_runs = sum(R,
+         na.rm = TRUE), mean_homeruns = mean(HR, na.rm = TRUE),
+     mean_runs = mean(R, na.rm = TRUE))
> head(team_summary)
```

```
  teamID yearID total_homeruns total_runs mean_homeruns mean_runs
1    ALT   1884              2         90     0.1176471  5.294118
2    ANA   1997            161        829     4.6000000 23.685714
3    ANA   1998            147        787     3.8684211 20.710526
4    ANA   1999            158        711     4.1578947 18.710526
5    ANA   2000            236        864     5.2444444 19.200000
6    ANA   2001            158        691     4.1578947 18.184211
```

4.3 Reshaping Data from Long to Wide

There are lots and lots of ways to organize your data. Some of this will depend on your personal preference. However, sometimes the analyses you will want to do will require your data to be organized in a certain way. This is often the case when you have observations taken from multiple time points, as is the case with our team summary date. Let's take a look at the first 10 rows of our team summary:

```
> head(team_summary, n = 10)
```

```
   teamID yearID total_homeruns total_runs mean_homeruns mean_runs
1     ALT   1884              2         90     0.1176471  5.294118
2     ANA   1997            161        829     4.6000000 23.685714
3     ANA   1998            147        787     3.8684211 20.710526
4     ANA   1999            158        711     4.1578947 18.710526
5     ANA   2000            236        864     5.2444444 19.200000
6     ANA   2001            158        691     4.1578947 18.184211
7     ANA   2002            152        851     3.8000000 21.275000
8     ANA   2003            150        736     3.4883721 17.116279
9     ANA   2004            162        836     4.2631579 22.000000
10    ARI   1998            159        665     3.4565217 14.456522
```

Notice how we have several rows for each of the teams, one for each year. This is what we call a "long" format. It's pretty human-readable and makes sense to us to look at, but it doesn't really conform to the principles of "tidy" data and therefore can be a bit problematic for R to work with.

To make our data "tidy," we need to convert it from a long format to a wide format. In a wide format, each observation will have one (and only one) row. See how we have several rows for each team in our summary as it is now? When we convert to a wide format, each team will have only one row.

How can we accomplish this? In this case, we have to create some new columns that combine the year with the variables of interest. In other words, we'll have a column for each year, for each of the four variables. This means that we'll have a lot more columns than we did before, but a lot fewer rows. That's why we call this format wide, rather than long!

There are a couple ways we can do this. One is the reshape function. This function takes a few arguments:

- the data (for the purposes of this demo, I'm just going to take a subset of my team summary data, everything from 2010 and on)
- the timevar - this is the variable that creates the separate records for each individual observation. In this data frame, yearID is that time variable - a team has multiple records based on different yearIDs.)
- the idvar - this is the variable that contains the identifier for each of my unique subjects. In this case, that's teams, but it could also be a patient ID, a specimen ID, etc
- direction - whether I'm converting to wide or to long, as reshape can do both

```
> wide <- reshape(subset(team_summary, yearID >= 2010), timevar = "yearID",
+     idvar = "teamID", direction = "wide")
> head(wide)
```

```
    teamID total_homeruns.2010 total_runs.2010 mean_homeruns.2010
22     ARI                 180             713           3.750000
71     ATL                 139             738           3.232558
132    BAL                 133             613           2.955556
281    BOS                 211             818           3.981132
600    CHA                 177             752           4.425000
741    CHN                 149             685           3.386364
    mean_runs.2010 total_homeruns.2011 total_runs.2011 mean_homeruns.2011
22        14.85417                 172             731           3.372549
71        17.16279                 173             641           3.844444
132       13.62222                 191             708           3.820000
281       15.43396                 203             875           4.142857
600       18.80000                 154             654           3.666667
741       15.56818                 148             654           3.523810
    mean_runs.2011 total_homeruns.2012 total_runs.2012 mean_homeruns.2012
22        14.33333                 165             734           3.367347
71        14.24444                 149             700           3.634146
132       14.16000                 214             712           4.115385
281       17.85714                 165             734           2.946429
600       15.57143                 211             748           4.688889
741       15.57143                 137             613           2.584906
    mean_runs.2012 total_homeruns.2013 total_runs.2013 mean_homeruns.2013
22        14.97959                 130             685           2.954545
71        17.07317                 181             688           4.113636
132       13.69231                 212             745           4.076923
281       13.10714                 178             853           3.708333
600       16.62222                 148             598           3.148936
741       11.56604                 172             602           3.071429
    mean_runs.2013 total_homeruns.2014 total_runs.2014 mean_homeruns.2014
```

```
22        15.56818                 118             615          2.269231
71        15.63636                 123             573          3.153846
132       14.32692                 211             705          4.795455
281       17.77083                 123             634          2.236364
600       12.72340                 155             660          3.522727
741       10.75000                 157             614          3.270833
    mean_runs.2014
22        11.82692
71        14.69231
132       16.02273
281       11.52727
600       15.00000
741       12.79167
```

Now, as you can see, we have lots of columns, but each team only has one row.

What if I want to change it back to a long format? Reshape can do that as well. First, I need to get a list of the names in the wide format that are going to be converted to single variables in our long format.

```
> names(wide)
```

```
 [1] "teamID"              "total_homeruns.2010" "total_runs.2010"
 [4] "mean_homeruns.2010"  "mean_runs.2010"      "total_homeruns.2011"
 [7] "total_runs.2011"     "mean_homeruns.2011"  "mean_runs.2011"
[10] "total_homeruns.2012" "total_runs.2012"     "mean_homeruns.2012"
[13] "mean_runs.2012"      "total_homeruns.2013" "total_runs.2013"
[16] "mean_homeruns.2013"  "mean_runs.2013"      "total_homeruns.2014"
[19] "total_runs.2014"     "mean_homeruns.2014"  "mean_runs.2014"
```

If we take a look at the list of names in our wide data frame, we can see that those are variables 2-21. Let's assign that list of variables to a name to make it easier to call them when we're writing our new reshape function.

```
> vars <- names(wide)[2:21]
```

Now we can use the reshape function to convert this from a wide to a long format. In this application of the function, we need to pass a few different arguments:

- the data
- varying - the names of the variables we're going to be breaking apart. I assigned the full list to the names vars, so I can just use that as a short cut
- the idvar - same as before
- direction - now I'm going to long

A note - reshape will try to guess how to break up the varying variables. By default, it expects to see the format of varname.time, like mean_ runs.2010. If this is NOT how your variables are named, you can specify what the separator is by using the "sep =" argument. For example, if you used an underscore between your variable name and time element, you'd pass the argument sep = "_ "

```
> long <- reshape(wide, varying = vars, idvar = "teamID", direction = "long")
> head(long)
```

```
         teamID time total_homeruns total_runs mean_homeruns mean_runs
ARI.2010    ARI 2010            180        713      3.750000  14.85417
ATL.2010    ATL 2010            139        738      3.232558  17.16279
BAL.2010    BAL 2010            133        613      2.955556  13.62222
BOS.2010    BOS 2010            211        818      3.981132  15.43396
CHA.2010    CHA 2010            177        752      4.425000  18.80000
CHN.2010    CHN 2010            149        685      3.386364  15.56818
```

5 Continuing Your R Journey

This class has provided a basic introduction to the syntax and basic functionality of R, but there is a lot more that R can do. Fortunately there are lots of ways to get help as you try to troubleshoot or figure out how to do new things in R.

5.1 Recommended Books

These resources provide nice overview information to help you learn more about accomplishing general data tasks in R.

- Lander, Jared P. *R for Everyone*
- Matloff, Norman. *The Art of R Programming: A Tour of Statistical Software Design*
- Teetor, Paul. *The R Cookbook: Proven Recipes for Data Analysis, Statistics, and Graphics*
- Wickham, Hadley. *Advanced R*

5.2 Google is Your Friend

Running into an error message you don't understand? Trying to figure out what function you should use to accomplish what you're trying to do? Chances are good that someone else has had your exact question, has gotten it answered, and you can find that answer by searching Google. As you look through your search results, a couple sites to keep an eye out for:

- Stack Overflow (http://stackoverflow.com): site where people can ask questions and the community will answer. Sometimes people can get a bit mean if you ask questions that have been answered elsewhere, so definitely do some searching before you post a question.
- CRAN - The Comprehensive R Archive Network (http://cran.r-project.org): basically the official site of R. Lots of documentation is hosted there, so you can usually find thorough descriptions of functions and packages
- R Bloggers (http://www.r-bloggers.com): user-created tutorials on a variety of topics, usually outlining in detail various approaches for accomplishing tasks
- UCLA's Institute for Digital Research and Education (http://www.ats.ucla.edu/stat/r/): a fairly comprehensive set of tutorials with lots of examples

5.3 Getting Help from the NIH Library

The NIH Library is here to help with your data and R questions! Handouts, tutorials, and (hopefully soon) videos of previous sessions are all available at http://nihlibrary.campusguides.com/dataservices/. To request a tutorial or consultation, please contact Lisa Federer at lisa.federer@nih.gov or use the Ask a Question form at http://nihlibrary.nih.gov/Services/Pages/AskAQuestion.aspx.